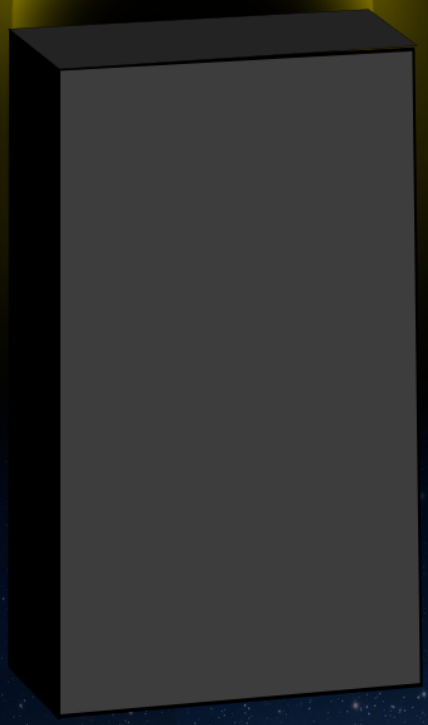
Clojure



Karthikeyan A K

Clojure

Karthikeyan.A.K

Table of Contents

Copyright . . . 1
Author . . . 2
Prerequisite . . . 3
Why this book? . . . 4
Where to get help . . . 5
  Local communities . . . 5
  Clojure website . . . 5
  Clojure forum . . . 5
  Reddit . . . 5
Getting this book . . . 6
1. Installing Stuff . . . 7
  1.1. Clojure . . . 7
  1.2. Leningen . . . 7
  1.3. Configuring IDE . . . 8
2. Hosted Language . . . 11
3. First Steps . . . 12
  3.1. REPL . . . 12
  3.2. Printing Things . . . 16
  3.3. Arithmetic . . . 17
  3.4. Types of Numbers . . . 21
  3.5. Strings . . . 24
  3.6. Variables . . . 25
  3.7. Clojure in file . . . 28
4. Data Structures . . . 29
  4.1. Vectors . . . 29
  4.2. Lists . . . 32
  4.3. Sets . . . 35
  4.4. Maps . . . 39
5. Reading Clojure Documentation . . . 43
  5.1. Clojure docs online . . . 43
6. Logic and comparison . . . 46
7. Conditions and branching . . . 51
  7.1. if . . . 51
  7.2. when . . . 51
  7.3. cond . . . 52
8. Functions . . . 53
  8.1. Recursion . . . 55
  8.2. Multimethods . . . 56

8.3. Pre and Post Condition Checking . . . 57
8.4. Docstring . . . 57
9. What is Clojure code . . . 60
10. Sequence . . . 61
11. Lazy Sequence . . . 62
11.1. repeat . . . 62
11.2. cycle . . . 62
11.3. iterate . . . 63
12. map, filter, reduce . . . 64
12.1. map . . . 64
12.2. filter . . . 64
12.3. reduce . . . 65
13. Destructuring . . . 66
Bibliography . . . 69

Copyright

This book is released under GFDL https://www.gnu.org/licenses/fdl-1.3.html.

Author

This book is written by me, Karthikeyan.A.K. You can write to me at mindaslab@protonmail.com, or text me in WhatsApp, Signal or Telegram @ +91 8428050777.

Prerequisite

There is one thing though, you must be familiar with using computers. I would suggest one to be familiar with Ubuntu GNU/Linux, you can get it here https://ubuntu.com. Install it on a computer, learn to use it, and get familiar with GNU/Linux. To get familiar with GNU/Linux one may visit and learn from http://linuxjourney.com.

Why this book?

For unknown reason very smart programmers are using Clojure, I haven't figured out why. The writing of this book is to figure out why. My last job was in a medical coding company where we maintained 3 Rails app, I got to code in two of them, one had 0.1+ million lines of code, was headed by a guy who knew how to code and was maintained okay, other one had 0.2+ million lines of code, had no test, headed by a guy who did not know how to code but could just do politics, that project prior to him was headed by people who did not know how to write maintainable software and senior developers in the project had to bear the brunt of history. It showed me how screwed Rails app can get if left unattended, so I want to search for something that's more idiot-proof.

When I first met Ruby on Rails, I was very sure PHP will be the king of web page making, and it's true even today, but Rails changed my life and made me more productive. I will not say that with Rails we cannot handle massive projects, in my present job extremely skilled people and me are doing just that, but then this book is just a search, a very similar search I did in my PHP days when I got a hint of the power of Ruby, and this book is to document my search, if Clojure is good enough or not.

Another reason why I am going into Clojure is because of Clojure Script. If I can use the same language on both client and server side and if its not dreaded JavaScript, who wouldn't want to explore such language? Lisp seems to be a very old language and Clojure seems to be a Lisp dialect. Somehow the inventors of Lisp seem to have struck the right balance early on, and I need to find out what it is.

I have lots of Data Science ideas, and looks like Clojure is fast and powerful enough to be used in those fields too. I want to explore that too, though I wonder if JVM ever runs on GPU for fast computation.

One negative I find is JVM. Java is controlled by Oracle, and it's an evil enterprise like Microsoft and Apple. That's the only negative I have, but let's see how things go.

Where to get help

If you are new to programming, or even if you are a seasoned programmer, chances are you could be stuck and may need help from time to time. For Clojure people these channels listed in this chapter may help.

Local communities

Its better to search for local Clojure and GNU/Linux communities and become friends with them. If you can find one, consider joining them. Those are the best ways to get help. One may look here https://clojure.org/community/user_groups for nearby group.

Clojure website

The Clojure website may be a good starting point to browse for help if you have the time and patience. One may visit it here https://clojure.org/. I am mentioning this because it is very possible that one may find some good piece of help I have left out.

Clojure forum

The official Clojure forum is here https://ask.clojure.org/, one may join it and start asking questions. Some of my initial doubts were cleared by this forum.

Clojure has a second forum here https://clojureverse.org/, it's called Clojureverse this one too seems to be popular.

Reddit

Reddit also has a Clojure community. If you are a reddit user, one may find the community here https://www.reddit.com/r/Clojure/.

Getting this book

One can get this book here https://clojure-book.gitlab.io/.

Chapter 1. Installing Stuff

1.1. Clojure

Clojure works on all OS platforms, you can install Clojure following the link here. Once done it's time for us to check the Clojure REPL. To know what is REPL, checkout https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop. Type `clj` in your terminal, and you must get something like this:

```
$ clj
Clojure 1.10.2
user=>
```

That's the Clojure REPL. Now type `(println "Hello World")` in it as shown below and press `ENTER`

```
$ clj
Clojure 1.10.2
user=> (println "Hello World")
```

You will see `Hello World` printed out as shown:

```
$ clj
Clojure 1.10.2
user=> (println "Hello World")
Hello World
nil
user=>
```

There is also a `nil` after the `Hello World`, don't worry much about that, we will see what is that later. After the nil you see once again `user⇒` which means Cloure is waiting for your type another command for it to execute.

If you want to come out of this Clojure REPL, type `CTRL` + `D`. You will get back to normal GNU/Linux prompt.

1.2. Leningen

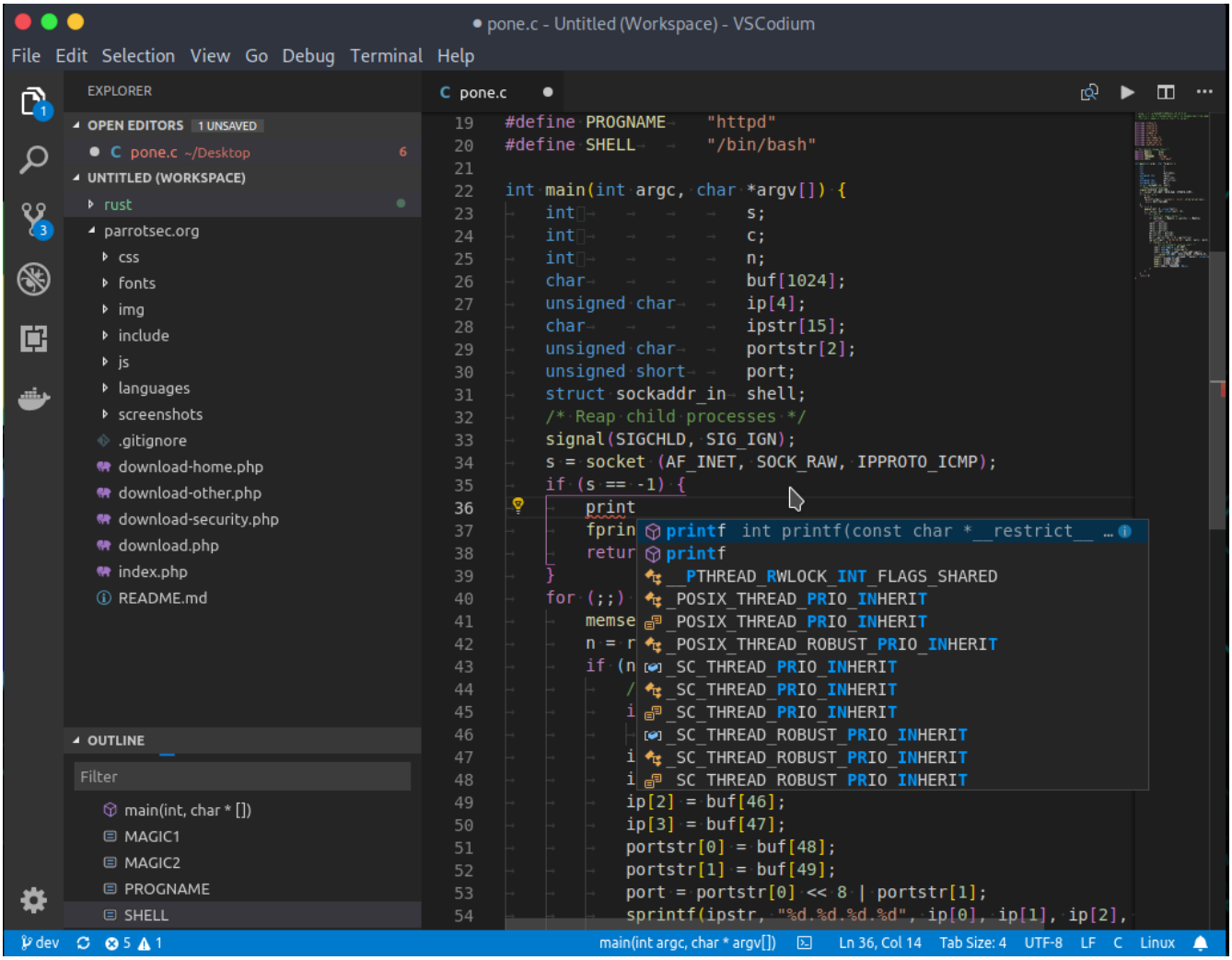
Leiningen is a tool that automates some stuff like setting up a Clojure project. Please refer to its website https://leiningen.org/ and install it. Once done type `lein repl` in your terminal, and you will see this:

```
$ lein repl
nREPL server started on port 64285 on host 127.0.0.1 - nrepl://127.0.0.1:64285
REPL-y 0.4.4, nREPL 0.8.3
Clojure 1.10.1
OpenJDK 64-Bit Server VM 1.8.0_152-release-1056-b12
    Docs: (doc function-name-here)
          (find-doc "part-of-name-here")
  Source: (source function-name-here)
 Javadoc: (javadoc java-object-or-class-here)
    Exit: Control+D or (exit) or (quit)
 Results: Stored in vars *1, *2, *3, an exception in *e

user=>
```

Press `CTRL` + `D` and it should stop. All is well, and you have installed Leningen.

1.3. Configuring IDE

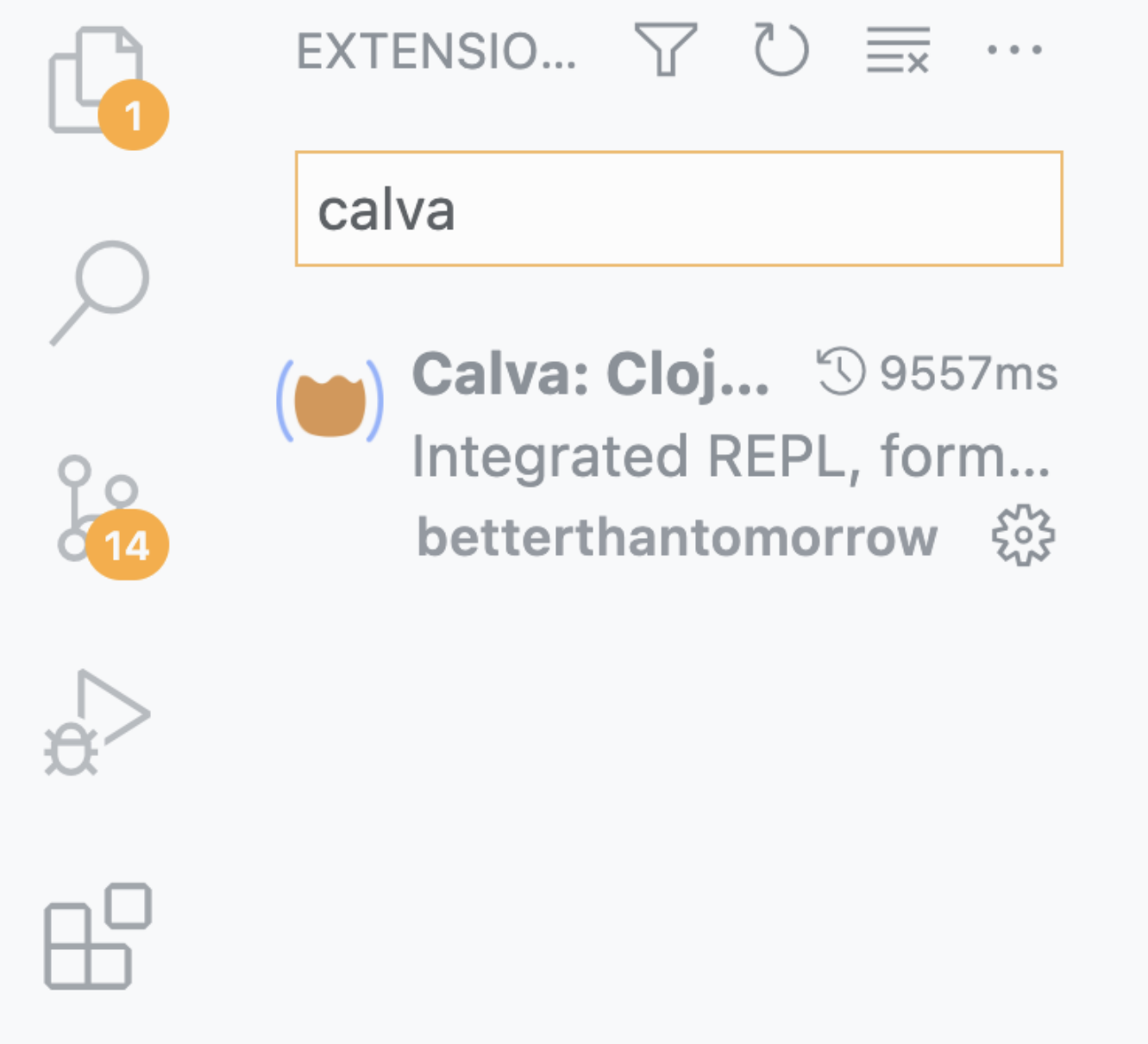
1.3.1. Installing VSCodium



You could use any IDE, but since this is my book, I would take the luxury of suggesting VSCodium. This IDE take the editor made by evil Microsoft and removes the bad part out of it. You could get the IDE here https://vscodium.com/ where the installation instructions are given.

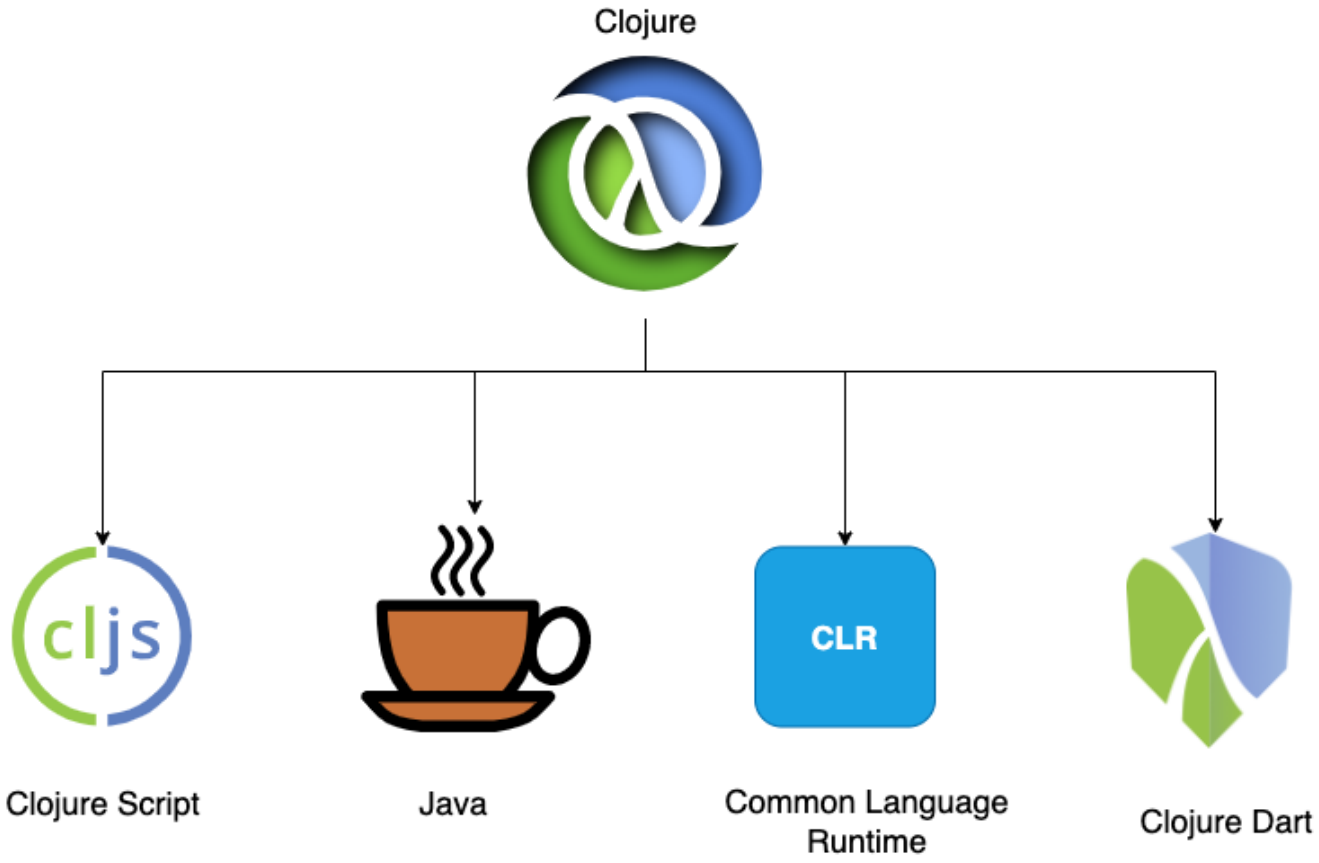
1.3.2. Calva

To use Clojure in VSCodium, there is an excellent plugin called Calva. You may learn about it here https://calva.io. Go to VSCodium extensions and install it by searching for it:



One may get started with Calva with excellent guides and video's here https://calva.io/getting-started/.

Chapter 2. Hosted Language



Clojure is a hosted language, which means its author does not intend it to be compiled to machine code that directly runs on a processor or an operating system. Clojure was initially intended to run on JRE (Java Runtime Environment), and gets converted to byte code. Since Clojure is hosted people found ways to get Clojure to get compiled to JavaScript, hence was born Clojure Script https://clojurescript.org/. Clojure can be compiled to Common Language Runtime https://clojure.org/about/clojureclr, the same thing what Dot Net family of languages gets compiled too. Clojure also gets compiled to Dart https://github.com/Tensegritics/ClojureDart. So by knowing Clojure one could target these platforms.

One should note that Clojure never promised WORA (write once run anywhere) like Java does, instead you need to change your code when you write Clojure code targeting different platforms. What you get is uniform Clojure Syntax everywhere.

Being a web developer I need can write backend in any beautiful language I like, but for the front end I am forced to used JavaScript. But in Clojure you can write both front and back end with Clojure without much cognitive load that come due to language switching.

Chapter 3. First Steps

3.1. REPL

When I first started to study Clojure, I was introduced to REPL driven development. I did not get it what it was during my early day, even if you don't get it, worry not, you will eventually get it. In this let's get a taste of REPL driven development.

3.1.1. REPL in Terminal

In your terminal type `clj` and you will be presented with something as shown

```
$ clj
Clojure 1.11.1
user=>
```

The `Clojure 1.11.1` tells us that Clojure version 1.11.1 is installed on my machine, and then it shows something like `user⇒`. That is Clojure is prompting you to enter something. Type in `(println "Hello World")` as shown and press enter:

```
$ clj
Clojure 1.11.1
user=> (println "Hello World")
Hello World
nil
user=>
```

You get the output `Hello World` printed out, and a strange thing `nil` in a line after that and once again you have been prompted to enter something at `user⇒`

You can now press `CTRL` + `D` to exit the REPL.

So what really happened when you typed `clj`. The first thing was Clojure prompted you with `user⇒`. You typed in something into the prompt, and pressed entered, Clojure REPL **R**ead it and **E**valuated it, understood that you want to print `Hello World`, next Clojure printed out the `Hello World`, then it also **P**rint a thing called `nil`, then it **L**ooped back again and prompted you with `user⇒`.

3.1.2. REPL in VSCodium

We have seen REPL in action in terminal, let's now see it in action in VSCodium. Create a folder named `code` and open that location with VSCodium, now crate a file named `hello_world.clj` in it with the following content:

hello_world.clj

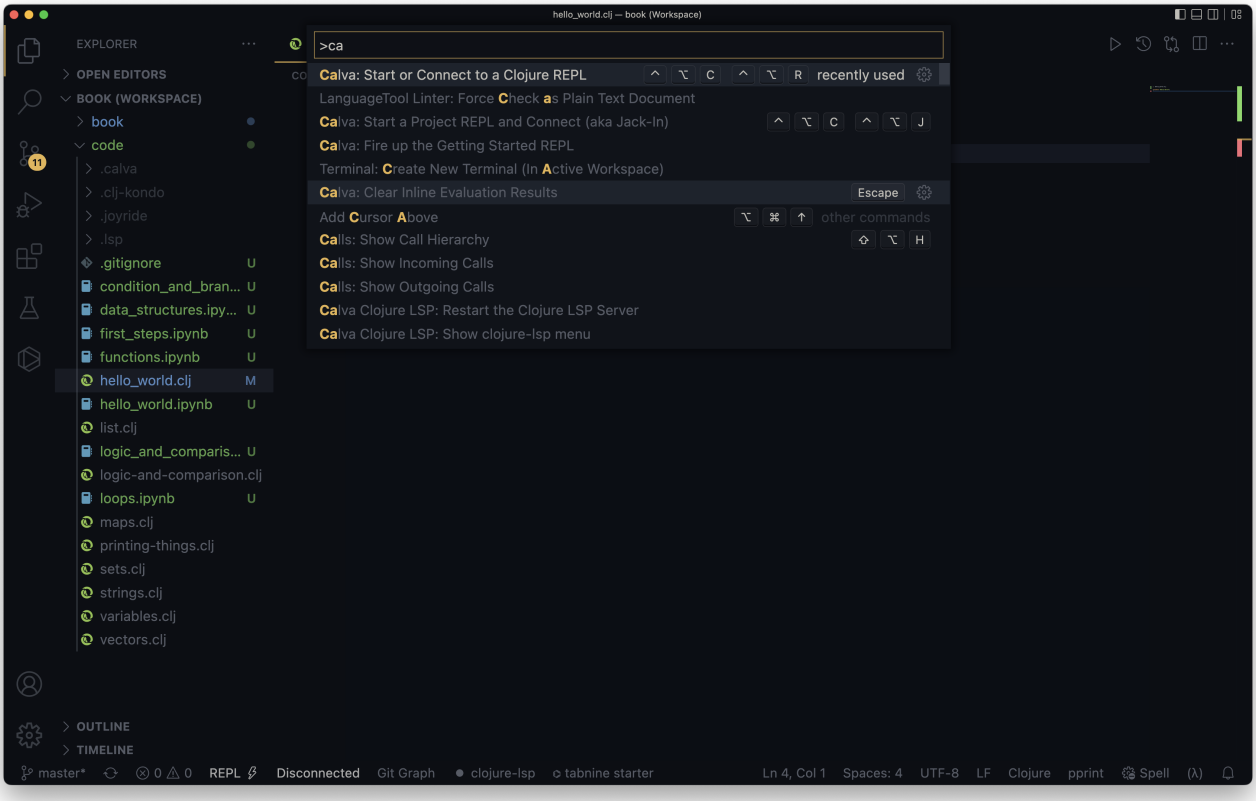
```
;; hello_world.clj
```

```
(println "Hello World")
```

Now in terminal `cd` into `code/` and type `lein repl`

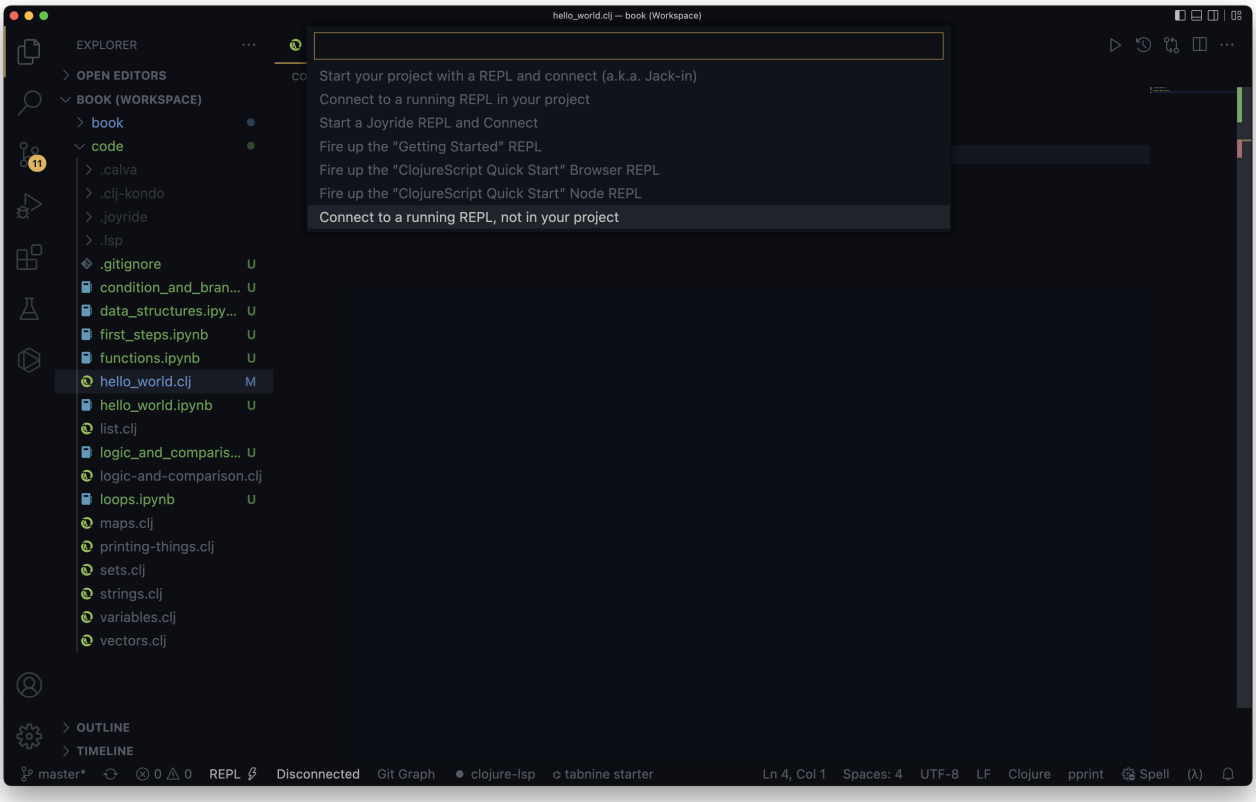
```
$ lein repl
nREPL server started on port 52897 on host 127.0.0.1 - nrepl://127.0.0.1:52897
REPL-y 0.4.4, nREPL 0.8.3
Clojure 1.10.1
OpenJDK 64-Bit Server VM 1.8.0_152-release-1056-b12
    Docs: (doc function-name-here)
          (find-doc "part-of-name-here")
  Source: (source function-name-here)
 Javadoc: (javadoc java-object-or-class-here)
    Exit: Control+D or (exit) or (quit)
 Results: Stored in vars *1, *2, *3, an exception in *e

user=>
```

The lein is short for Leningen, if you might remember installing it and checking if it launched. `repl` tells `lein` to start a REPL. You will see a output as shown. Note `nrepl://127.0.0.1:52897` in the above output, it means that there is a REPL server running at IP 127.0.0.1 [1] and at port number [2] 52897. Note down `127.0.0.1:52897`. May be the port number could vary in your case.

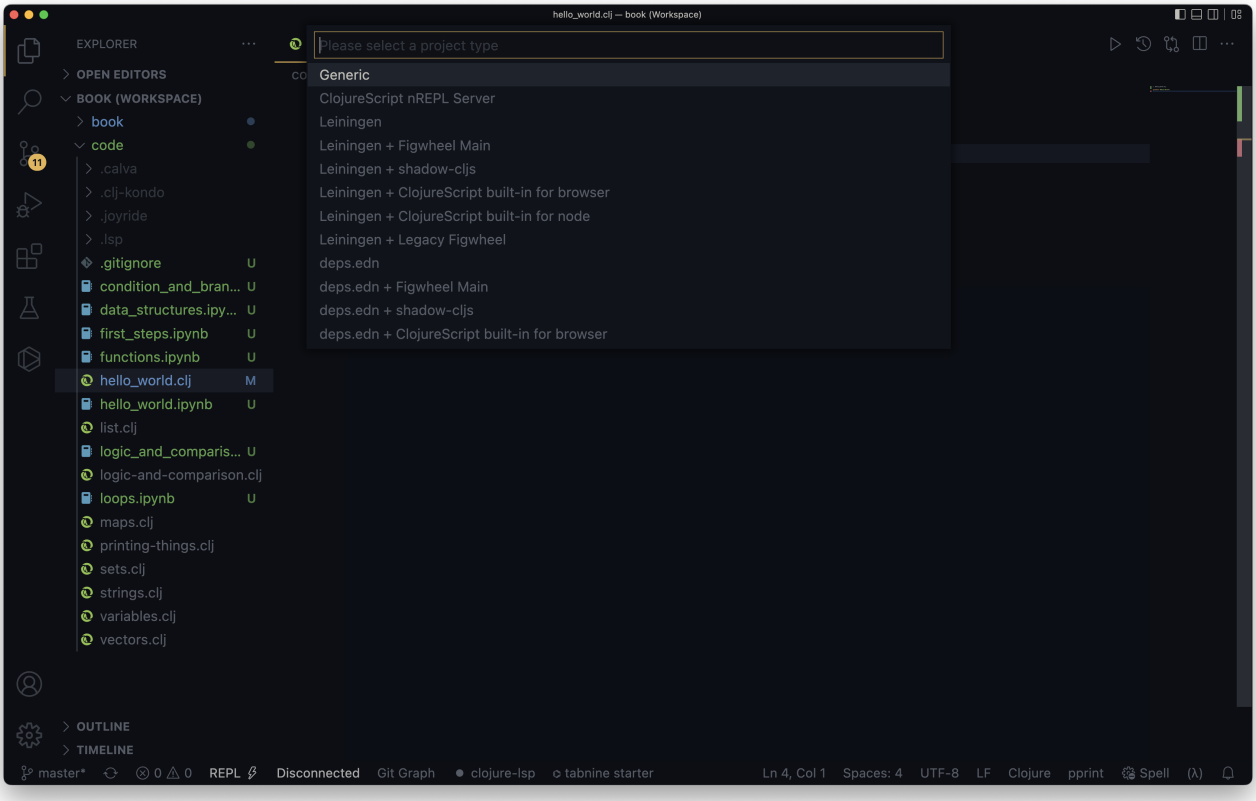


In VSCodium type `CTRL` + `P` or `command` + `P` and type `>`, now you can type in some commands for VSCodium to execute. In it try typing `Calva: Start or connect to a Clojure REPL`, before you finish

typing it, VSCodium would have selected that option for you, press `ENTER` or click on it.

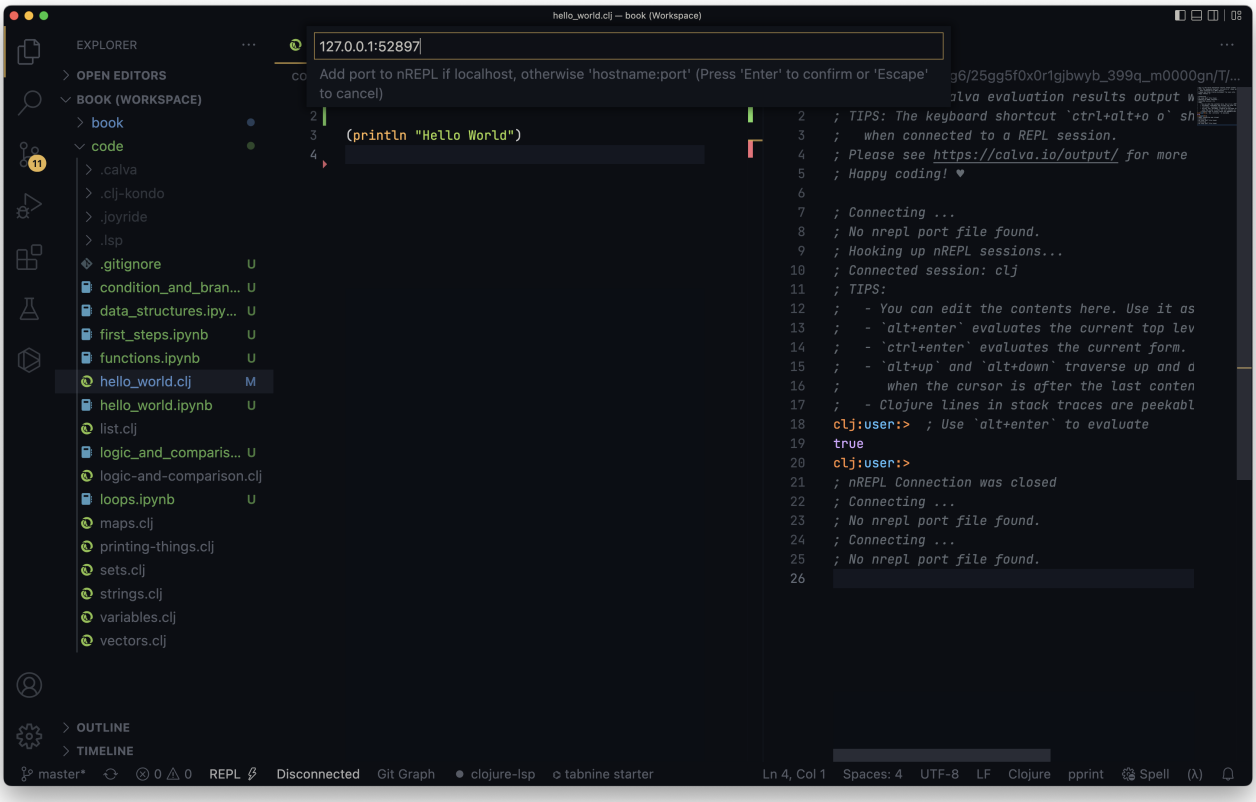


VSCodium will present another set of menus, in it select `Connect to a running REPL, not in your project`.

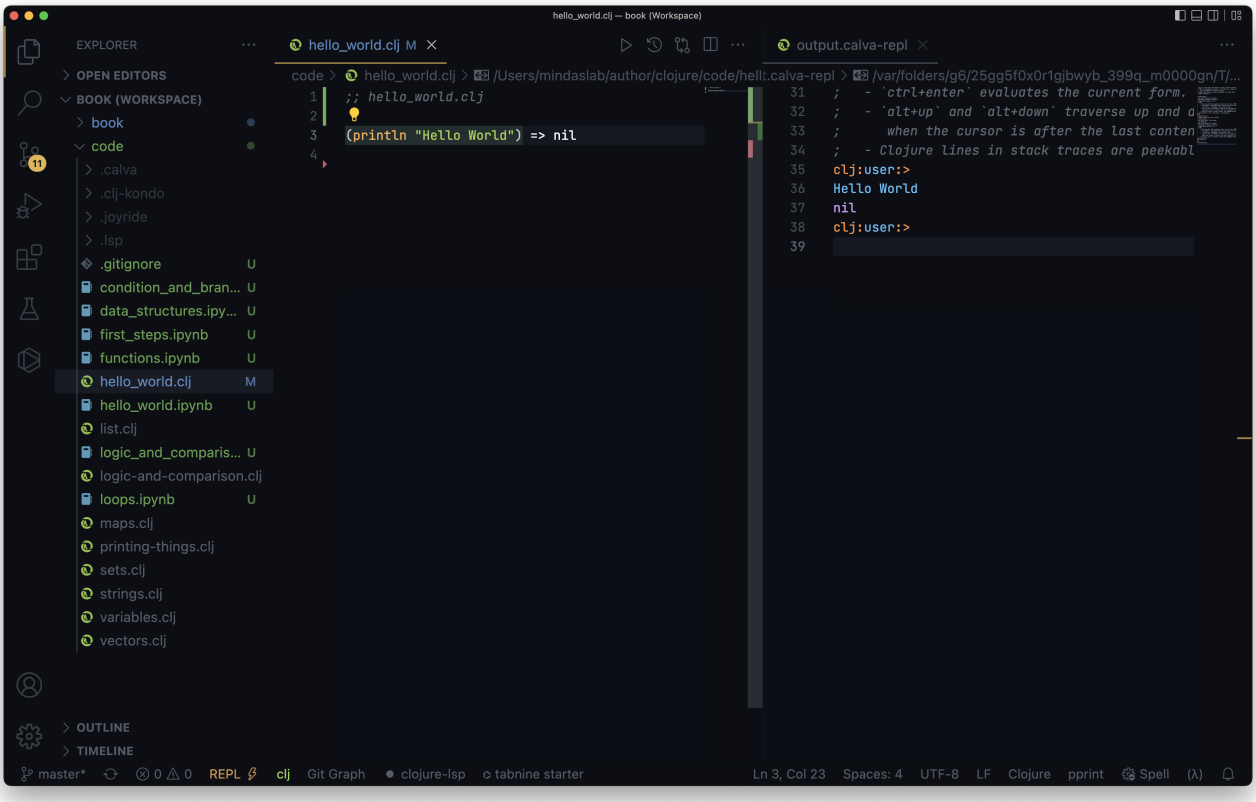


Now in the presented option select `Generic` as shown above. Now it would prompt for the IP and

port number where your REPL is running.



Paste the copied value `127.0.0.1:52897` and press enter. At the right you will see a Clojure REPl window being opened. Now in the file at left in `hello_world.clj`, keep your cursor inside `(println "Hello World")` and press `ALT` + `ENTER` or `option` + `return`



You will see the `(println "Hello World")` evaluated and printed in right-hand REPL pane. Congratulations, you learned how to connect REPL with VSCodium.

3.2. Printing Things

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/printing_things.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

First let's print `Hello World!`, copy the stuff below, paste it and execute it:

```
(println "Hello World!")
```

Output

```
Hello World!
nil
```

So you get `Hello World!` as well as a `nil` printed in the next line. Forget about the `nil`, we will look into it when we see functions, but our mission is accomplished.

Commenting is very essential in coding. Comments are nothing but notes for the developer who is reading the code, while executing the code, the computer will ignore comments. In clojure everything that follows after semicolon `;` is a comment. So in the code below:

```
(println "Hello World!") ; Says Hello to this world
```

Output

```
Hello World!
nil
```

`; Says Hello to this world` is a comment which the Clojure interpreter will ignore. In the code above, we are commenting in the same line as there is code, usually its a convention to use just one semicolon `;` for such things. IF you want an entire line dedicated for comment, we use two semicolons `;;` as shown below:

```
;; This program says helo to this world
(println "Hello World!")
```

Output

```
Hello World!
```

```
nil
```

Both single and double semicolons makes no difference, but that the way conventions have evolved in Clojure.

We have printed `Hello World`, but then wht if we ant to print something else with it. With `println` its simple, just give other stuff too and it will print as shown:

```
(println "Hello World!" "Try staying cool.")
```

Output

```
Hello World! Try staying cool.
nil
```

So you have successfully printed something in Clojure.

3.3. Arithmetic

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/arithmetic.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

In this section let's see about Math in Clojure. First let's add two numbers 40 and 2, it been done as shown:

```
(+ 40 2)
```

Output

```
42
```

In the above thing we have the plus `+` sign, and we pass 40 and 2 to it for it to add and return 42. Similarly let's see what will happen when we pass 44 and 2 to minus `-` sign:

```
(- 44 2)
```

Output

```
42
```

It seems to subtract the second argument 2 from 44 and returns 42.

Now let's see what happens when we pass 6 and 7 to asterisk * or a a star:

```
(* 6 7)
```

Output

```
42
```

We see it returns 42, which is a multiple of 6 and 7.

Now let's see what happens when we pass 210 and 5 to slash /:

```
(/ 210 5)
```

Output

```
42
```

Looks like 210 is divided by 5 and we get 42.

Now let's see what will happen if we pass more than 2 numbers to plus +:

```
(+ 1 2 3 4 5)
```

Output

```
15
```

All are added and the result of addition is got.

Now look at the code below and can you figure out how it works?

```
(/ (+ 1 2 3 4 5) 5)
```

Output

```
3
```

Let's take the code `(/ (+ 1 2 3 4 5) 5)`, concentrate on the inner most braces `(+ 1 2 3 4 5)`, here a bunch of numbers are been passed to + and we should get the sum of 1 to 5, that is 15 returned. So in place of `(+ 1 2 3 4 5)` let's substitute 15, so we get a expression like this `(/ 15 5)`, which is

nothing but 15 and 5 are passed to `/`, which means 15 divided by 5, and hence we get 3.

Now let's pass one or more argument to `/`, passing 1, 2 and 3 to it, we get $\frac{1}{6}$ as output as shown below:

```
(/ 1 2 3) ; this gives out a ratio, a clojure data type
```

Output

```
1/6
```

This is a fraction, to get a decimal output of a division, append a `.0` to the number when passing to `/`

```
(/ 1 6.0)
```

Output

```
0.16666666666666666
```

If you don't understand it now, don't worry, as you read this book it will get clarified.

It's enough that any one number that's been passed to `/` has a decimal value that is point `.` something appended to it, we will still get decimal rather than an fraction as output as shown below:

```
(/ 1 2.0 3)
```

Output

```
0.16666666666666666
```

Sme is the case below too:

```
(/ 1 2 3.0)
```

Output

```
0.16666666666666666
```

Now if we need to get quotient of a division, we can pass numbers to a function named `quot` as

shown:

```
(quot 14 5)
```

Output

```
2
```

To get reminder of a division operation, pass numbers to rem function as shown:

```
(rem 14 5)
```

Output

```
4
```

Let's now try to find out area of a circle using clojure. We know the area is πr^2, where π is roughly $\frac{22}{7}$. So let's see how to code it. Look at the code below:

```
;; Let's calculate area of circle of radius 7 units
(* (/ 22 7) (* 7 7))
```

Output

```
154N
```

Here we represent π by `(/ 22 7)` the r^2 where r 7 is `(* 7 7)`, this is multiplied with `(/ 22 7)` and we get `(* (/ 22 7) (* 7 7))`. Now the output we got is `154N`, I have no idea what the `N` is.

If I add a `.0` to any number below, the annoying `N` disappears:

```
(* (/ 22 7) (* 7 7.0))
```

Output

```
154.0
```

Rather than coding our own π as `(/ 22 7)`, since Clojure is built on Java, we can tap into Java `Math` library (https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html), and use it's methods. This is one advantage of Clojure, since it's a hosted language, it can take advange of libraries provided by the host platform. So we replace `(/ 22 7)` with `Math/PI` as shown below:

```
(* Math/PI (* 7 7))
```

Output

```
153.93804002589985
```

Similarly to compute x^y, we can use `Math/pow` so instead of `(* 7 7)`, we can replace it with `(Math/pow 7 2)` as shown below:

```
(* Math/PI (Math/pow 7 2))
```

Output

```
153.93804002589985
```

3.4. Types of Numbers

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/types_of_numbers.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

There are different types of numbers in computing just like there are different types of numbers in Mathematics. For example whole numbers are called as Long in Clojure. You can verify it by passing a whole number to a function named `type` which tells the type of data been passes to it as shown below:

```
(type 147)
```

Output

```
java.lang.Long
```

The numbers with decimal value are called as Double in clojure as you can see below:

```
(type 147.67)
```

Output

```
java.lang.Double
```

The value of π too is a double in Clojure:

```
(type Math/PI)
```

Output

```
java.lang.Double
```

If two whole numbers or Long divides another Long, what you get is a Ratio in Clojure:

```
(type (/ 84 32))
```

Output

```
clojure.lang.Ratio
```

As you can see below $\frac{84}{32}$ in Clojure does not provide a decimal value, but outputs a Ratio:

```
(/ 84 32)
```

Output

```
21/8
```

You can convert one type of number to another, for example 32 is of type Long:

```
(type 32)
```

Output

```
java.lang.Long
```

But you can convert it to double by passing it to a `double` function as shown:

```
(double 32)
```

Output

```
32.0
```

When querying about it's type, it does say Double:

```
(type (double 32))
```

Output

```
java.lang.Double
```

Say you don't want a division of two Long's to give you a Ratio as output, to prevent it, convert any one of the number thats been passed to `/` function into a Double as shown below:

```
(/ 84 (double 32))
```

Output

```
2.625
```

The output you get is a Double and not a Ratio:

```
(type (/ 84 (double 32)))
```

Output

```
java.lang.Double
```

If you wan to just get a non decimal part of a Double, pass that double to long function as shown:

```
(long 42.32)
```

Output

```
42
```

As you see, when a Double is passed to `long` function, it gets type casted to Long:

```
(type (long 42.32))
```

Output

```
java.lang.Long
```

3.5. Strings

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/strings.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

In this book I think your very first clojure program would have been this:

```
(println "Hello World!")
```

It prints `Hello World` on the screen. The `"Hello World!"` which is enclosed by double quoted `"` is nothing but a string of characters and hence its called a string.

anything that's enclosed by double quotes in Clojure is a String as you can see below:

```
(type "Hello")
```

Output

```
java.lang.String
```

Strings can be joined with even non String things using `str` function as shown:

```
(str "1 + 2 is " (+ 1 2))
```

Output

```
"1 + 2 is 3"
```

In the above code `(+ 1 2)` returns Long, but it can be joined with a string `"1 + 2 is "` by passing them both to `str` function.

You can get the character count in a string using using the `count` function as shown below:

```
(count "Hello")
```

Output

```
5
```

In the code below, to `str` function we are passing 6 arguments. the first five are numbers , and the last one is a string `" and so on⋯"`:

```
(str 1 2 3 4 5 " and so on...")
```

Output

```
"12345 and so on..."
```

You can see how Clojure neatly concatenates them and returns out a string.

Clojure has a inbuilt String library, you can read it's documentation here https://clojuredocs.org/clojure.string. From it let's use the function `reverse` to reverse a String:

```
(clojure.string/reverse "Hello")
```

Output

```
"olleH"
```

Now let's covert all characters to upper case:

```
(clojure.string/upper-case "Hello")
```

Output

```
"HELLO"
```

I hope this section has briefly explained something about strings. You wil learn more as you go by. Hang on....

3.6. Variables

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/variables.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

Imagine variables as boxes containing some value, this can then be used in code to make it more human readable. For example look at the code below:

```
(def my-name "Karthik")
```

Output

```
#'user/my-name
```

I have assigned String "Karthik" to a variable named `my-name`, now I can use this `my-name` any where in the file. If you look at creation of a variable, you can see these three things, first is of course empty braces:

```
 ()
```

Then we have the `def` function:

```
(def)
```

Following the `def` we have the variable name, in this case its `my-name`:

```
(def my-name)
```

As a second argument to `def`, we give the value that needs to be stored in `my-name` that is `"Karthik"` in this case:

```
(def my-name "Karthik")
```

So that is how we define a variable. Usually the variable defined with `def` is called a global variable, that is it's available any where in the program. If it doesn't make sense now, don't worry.

Now we can print this variable as shown:

```
(println my-name)
```

Output

```
Karthik
nil
```

Over here I am adding `"Hello"` before the variable `my-name` and its as equivalent of using string `"Karthik"` instead of `my-name`:

```
(println "Hello" my-name)
```

Output

```
Hello Karthik
nil
```

Look at the code below:

```
(def greeting (str "Hello " my-name "!"))
(println greeting)
```

Output

```
Hello Karthik!
nil
```

I am having a variable called `greeting` defined above and its been assigned the result of `(str "Hello " my-name "!")`. That is three strings `"Hello "`, `my-name` and `"!"` are joined together and the result is stored in `greeting`. Finally we print it using `(println greeting)`.

Variables like `my-name` and `greeting` can be used any where in code, because they are defined using `def` keyword. If you want a localized variable inside a code block `()`, you can use `let` to define it as shown:

```
(let [local-variable "something"]
  (println local-variable))
```

Output

```
something
```

In the above code, we define a variable named `local-variable` to be `"something"`, and we successfully print it, but outside the bounding braces `(let ⋯)`, this local variable fails to exist as you can see from the code below:

```
(println local-variable)
```

Output

```
; Syntax error compiling at (variables.clj:16:1).
```

```
; Unable to resolve symbol: local-variable in this context
```

Trying to print `local-variable` results in an error.

3.7. Clojure in file

Till now you might have run your code on REPL, either in your terminal or in your IDE, possibly VSCodium. Now let's see how to store your code in a file and run it. First type the following name in a text editor:

hello_world.clj

```
;; hello_world.clj

(println "Hello World")
```

Save the code as `hello_world.clj`, in your terminal `cd` to that directory, then type this in terminal:

```
$ clj hello_world.clj
```

The entire program will be run and you will see `Hello World` as output. I got this as output:

```
$ clj hello_world.clj
WARNING: Implicit use of clojure.main with options is deprecated, use -M
Hello World
```

[1] https://en.wikipedia.org/wiki/Localhost

[2] https://en.wikipedia.org/wiki/Port_(computer_networking)

Chapter 4. Data Structures

Almost every programming language offers some way to pack, organize and access data in computer's RAM [1], this eases the burden out of programmers from writing lot of code to manage data stored in computer's temporary memory.

String according to me is a data structure too. It enables us to represent any written language in computers memory, and helps us perform operations with it. In this section we are going to see about other data structures Clojure provides which you can use it out of the box.

4.1. Vectors

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/vectors.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

Imagine racks with lot of compartments in it, the first compartment is labelled 0, the second 1 and so on. You can put something into this rack and get it back if you know the rack number aka the index. This is a kind of real world analog to vectors.

Look at the code below:

```
[1 2 3 4] ; a very simple vector
```

Output

```
[1 2 3 4]
```

This is how you represent a vector in Clojure. It start's with a opening square braces `[` and ends with a closing one `]`, and any thing in between are the elements of the vector. In the above code you got 4 elements namely `1`, `2`, `3` and `4`.

Another way to create a vector is to pass stuff to a function called `vector` as shown below:

```
(vector 1 2 3)
```

Output

```
[1 2 3]
```

In the above code we have passed `1`, `2` and `3` to a the function vector and we have got out a vector `[1 2 3]`.

Till now we have seen creating vectors with numbers, in reality you can create vector with any type

of value:

```
[1 true "Bashir"] ; vector containing multiple data types
```

Output

```
[1 true "Bashir"]
```

In the above code we have created a three element vector with `1` which is a Long, `true` which is boolean (you will learn about them soon), and `"Bashir"` which is a String.

In the code below, we are defining a variable named `friends` and we are assigning a vector that contains four names to it:

```
(def friends
    ["Ram" "Bashir" "Antony" "Buddha"])
```

Output

```
#'user/friends
```

Now let's see what is the first element of the vector `friends` is, for that we pass `friends` to a function named `first`:

```
(first friends)
```

Output

```
"Ram"
```

and we get the first element in the vector.

There is a function called `rest` which when given a vector, omits the first element and returns the rest of the elements in the vector as shown belo:

```
(rest friends)
```

Output

```
("Bashir" "Antony" "Buddha")
```

As you see `"Ram"` is ignored as it's the first element and the rest is returned.

Let's check the tyoe of `friends`

```
(type friends)
```

Output

```
clojure.lang.PersistentVector
```

Fro the output it seems clear that it's a vector, but then let's checkout what type the `rest` returns:

```
(type (rest friends))
```

Output

```
clojure.lang.PersistentVector$ChunkedSeq
```

Though it seems to return `("Bashir" "Antony" "Buddha")`, which are not enclosed in square braces, on querying the type, it seems to be a chunk of the passed in vector.

We can access any element by it's index in a vector using the `nth` function who's first argument should be a vector and second should be the index number as shown:

```
(nth friends 3)
```

Output

```
"Buddha"
```

If you see, at index `3` of `friends` is a String `"Buddha"`.

The third index can also be access like shown:

```
(friends 3)
```

Output

```
"Buddha"
```

Doesn't it look like when you did `(def friends <a vector>)`, clojure seems to have constructed a

function called `friends` which can take a number, and the value at the index gets returned?

To add an element at the end of vector, use the function `conj` whcih takes in vector as first argument and the value to be added as the second:

```
(conj friends "Periyaar")
```

Output

```
["Ram" "Bashir" "Antony" "Buddha" "Periyaar"]
```

In the above code we have added `"Periyaar"` to friends, but this does not mean `friends` as been modified, we can print `friends`:

```
(println friends)
```

Output

```
[Ram Bashir Antony Buddha]
nil
```

and it still shows the old values without `"Periyaar"`, in reality Clojure constructs a new vector when we say `(conj friends "Periyaar")` and returns it out.

To add a element before a vector, use `cons` as shown below:

```
(cons "Periyaar" friends)
```

Output

```
("Periyaar" "Ram" "Bashir" "Antony" "Buddha")
```

Does it modify `friends`? Now research what is immutability [2] in computing.

4.1.1. Exercise

- How to get total number of elements in a vector?
- What will happen if I give `(friends 42)`?

4.2. Lists

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/

> master/list.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

```
'(1 2 3 4)
```

Output

```
(1 2 3 4)
```

```
(list 1 2 3)
```

Output

```
(1 2 3)
```

```
(list 1 2 3)
```

Output

```
(1 2 3)
```

```
(def friends-list
    '("Ram" "Bashir" "Antony" "Buddha"))
```

Output

```
#'user/friends-list
```

```
(count friends-list)
```

Output

```
4
```

```
(first friends-list)
```

Output

```
"Ram"
```

```
(rest friends-list)
```

Output

```
("Bashir" "Antony" "Buddha")
```

```
(nth friends-list 3)
```

Output

```
"Buddha"
```

```
(friends-list 3)
```

Output

```
Execution error (ClassCastException) at user/eval5554 (REPL:1).
clojure.lang.PersistentList cannot be cast to clojure.lang.IFn


       core.clj:  3214 clojure.core$eval/invokeStatic

       core.clj:  3210 clojure.core$eval/invoke

       main.clj:   437 clojure.main$repl$read_eval_print__9086$fn__9089/invoke

       main.clj:   458 clojure.main$repl$fn__9095/invoke

       main.clj:   368 clojure.main$repl/doInvoke

    RestFn.java:  1523 clojure.lang.RestFn/invoke

       AFn.java:    22 clojure.lang.AFn/run

       AFn.java:    22 clojure.lang.AFn/run

    Thread.java:   745 java.lang.Thread/run
```

```
(conj friends-list "Periyaar")
```

Output

```
("Periyaar" "Ram" "Bashir" "Antony" "Buddha")
```

4.3. Sets

https://gitlab.com/clojure-book/code/-/raw/master/sets.clj

```
#{1 2 3 4}
```

Output

```
#{1 4 3 2}
```

```
(type #{1 2 3 4})
```

Output

```
clojure.lang.PersistentHashSet
```

```
#{1 2 3 4 4}
```

Output

```
Syntax error reading source at (REPL:1:13).
Duplicate key: 4


    PersistentHashSet.java:    68 clojure.lang.PersistentHashSet/createWithCheck

            LispReader.java:  1366 clojure.lang.LispReader$SetReader/invoke

            LispReader.java:   853 clojure.lang.LispReader$DispatchReader/invoke

            LispReader.java:   285 clojure.lang.LispReader/read

                   core.clj:  3768 clojure.core$read/invokeStatic

                   core.clj:  3741 clojure.core$read/invoke

                   main.clj:   433
clojure.main$repl$read_eval_print__9086$fn__9087/invoke
```

```
                  main.clj:   432 clojure.main$repl$read_eval_print__9086/invoke
                  main.clj:   458 clojure.main$repl$fn__9095/invoke
                  main.clj:   368 clojure.main$repl/doInvoke
               RestFn.java:  1523 clojure.lang.RestFn/invoke
                  AFn.java:    22 clojure.lang.AFn/run
                  AFn.java:    22 clojure.lang.AFn/run
               Thread.java:   745 java.lang.Thread/run
```

```
#{"Apple" "Orange" "Mango" "Banana"}
```

Output

```
#{"Mango" "Orange" "Apple" "Banana"}
```

```
#{"Apple" "Orange" "Mango" "Banana" "Apple"}
```

Output

```
Syntax error reading source at (REPL:1:45).
Duplicate key: Apple


    PersistentHashSet.java:    68 clojure.lang.PersistentHashSet/createWithCheck
           LispReader.java:  1366 clojure.lang.LispReader$SetReader/invoke
           LispReader.java:   853 clojure.lang.LispReader$DispatchReader/invoke
           LispReader.java:   285 clojure.lang.LispReader/read
                  core.clj:  3768 clojure.core$read/invokeStatic
                  core.clj:  3741 clojure.core$read/invoke
                  main.clj:   433
clojure.main$repl$read_eval_print__9086$fn__9087/invoke
                  main.clj:   432 clojure.main$repl$read_eval_print__9086/invoke
```

```
                main.clj:   458 clojure.main$repl$fn__9095/invoke
                main.clj:   368 clojure.main$repl/doInvoke
              RestFn.java:  1523 clojure.lang.RestFn/invoke
                 AFn.java:    22 clojure.lang.AFn/run
                 AFn.java:    22 clojure.lang.AFn/run
              Thread.java:   745 java.lang.Thread/run
```

```
(def fruits #{"Apple" "Orange" "Mango" "Banana"})
```

Output

```
#'user/fruits
```

```
(contains? fruits "Banana")
```

Output

```
true
```

```
(contains? fruits "Jack Fruit")
```

Output

```
false
```

```
(fruits "Banana")
```

Output

```
"Banana"
```

```
(fruits "Jack Fruit")
```

Output

```
nil
```

```
("Banana" fruits)
```

Output

```
Execution error (ClassCastException) at user/eval5624 (REPL:1).
java.lang.String cannot be cast to clojure.lang.IFn


       core.clj:  3214 clojure.core$eval/invokeStatic

       core.clj:  3210 clojure.core$eval/invoke

       main.clj:   437 clojure.main$repl$read_eval_print__9086$fn__9089/invoke

       main.clj:   458 clojure.main$repl$fn__9095/invoke

       main.clj:   368 clojure.main$repl/doInvoke

    RestFn.java:  1523 clojure.lang.RestFn/invoke

       AFn.java:    22 clojure.lang.AFn/run

       AFn.java:    22 clojure.lang.AFn/run

    Thread.java:   745 java.lang.Thread/run
```

```
(def programming-languages #{:ruby :python :clojure})
```

Output

```
#'user/programming-languages
```

```
(contains? programming-languages :ruby)
```

Output

```
true
```

```
(contains? programming-languages :java)
```

Output ----false----

```
(programming-languages :ruby)
```

Output

```
:ruby
```

```
(:ruby programming-languages)
```

Output

```
:ruby
```

```
(conj programming-languages :perl)
```

Output

```
#{:clojure :python :perl :ruby}
```

```
(disj programming-languages :perl)
```

Output

```
#{:clojure :python :ruby}
```

4.4. Maps

https://gitlab.com/clojure-book/code/-/raw/master/maps.clj

```
{ "name" "Bashir" "age" 12 }
```

Output

```
{"name" "Bashir", "age" 12}
```

```
(hash-map "name" "Bashir"
```

```
          "age" 20)
```

Output

```
{"age" 20, "name" "Bashir"}
```

```
(def friend { "name" "Bashir" "age" 12 })
```

Output

```
#'user/friend
```

```
(get friend "name")
```

Output

```
"Bashir"
```

```
(friend "name")
```

Output

```
"Bashir"
```

```
(type "name")
```

Output

```
java.lang.String
```

```
(type :name)
```

Output

```
clojure.lang.Keyword
```

```
(def wise-friend {:name "Periyaar"
                  :age 90})
```

Output

```
#'user/wise-friend
```

```
(wise-friend :name)
```

Output

```
"Periyaar"
```

```
(get wise-friend :age)
```

Output

```
90
```

```
(print (wise-friend :name)
       "is very wise.")
```

Output

```
Periyaar is very wise.nil
```

```
(assoc wise-friend :belief "Rationalism")
```

Output

```
{:name "Periyaar", :age 90, :belief "Rationalism"}
```

```
(wise-friend :belief)
```

Output ----nil----

```
(wise-friend :age)
```

Output ----90----

```
(dissoc wise-friend :age)
```

Output

```
{:name "Periyaar"}
```

```
(keys wise-friend)
```

Output

```
(:name :age)
```

```
(vals wise-friend)
```

Output

```
("Periyaar" 90)
```

```
(assoc wise-friend "name" "Ramasamy")
```

Output

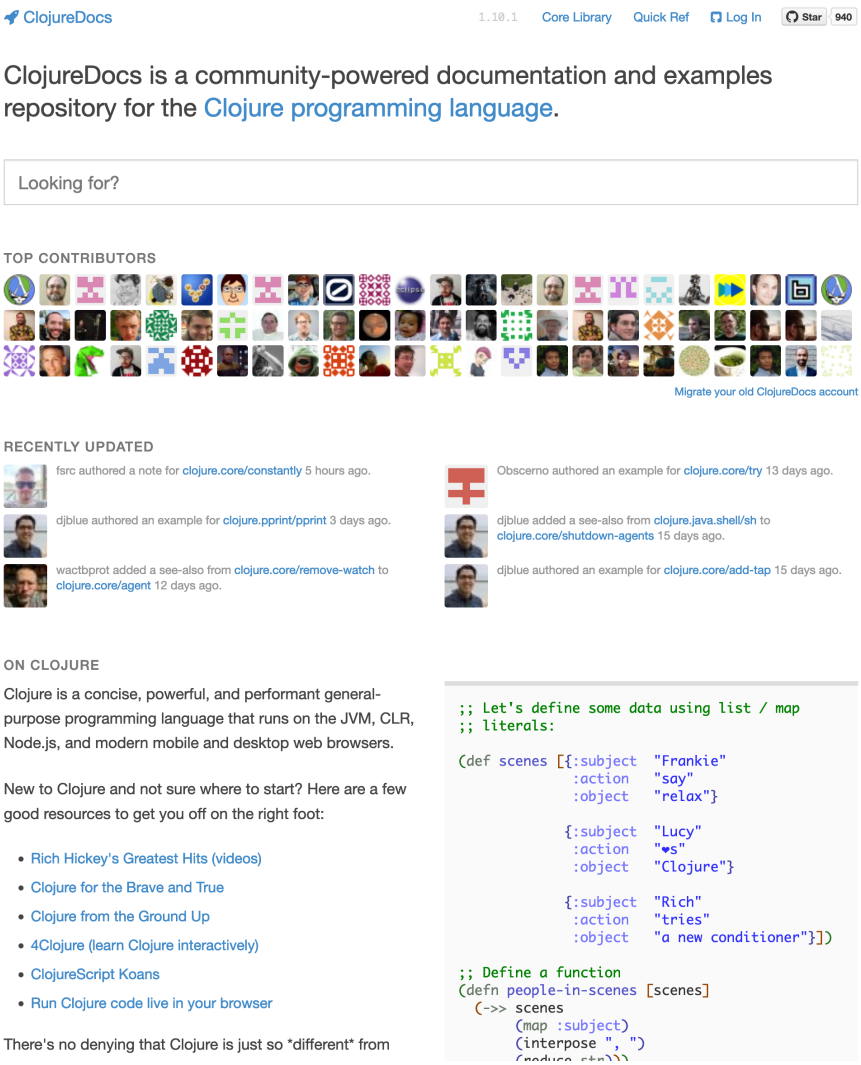
```
{:name "Periyaar", :age 90, "name" "Ramasamy"}
```

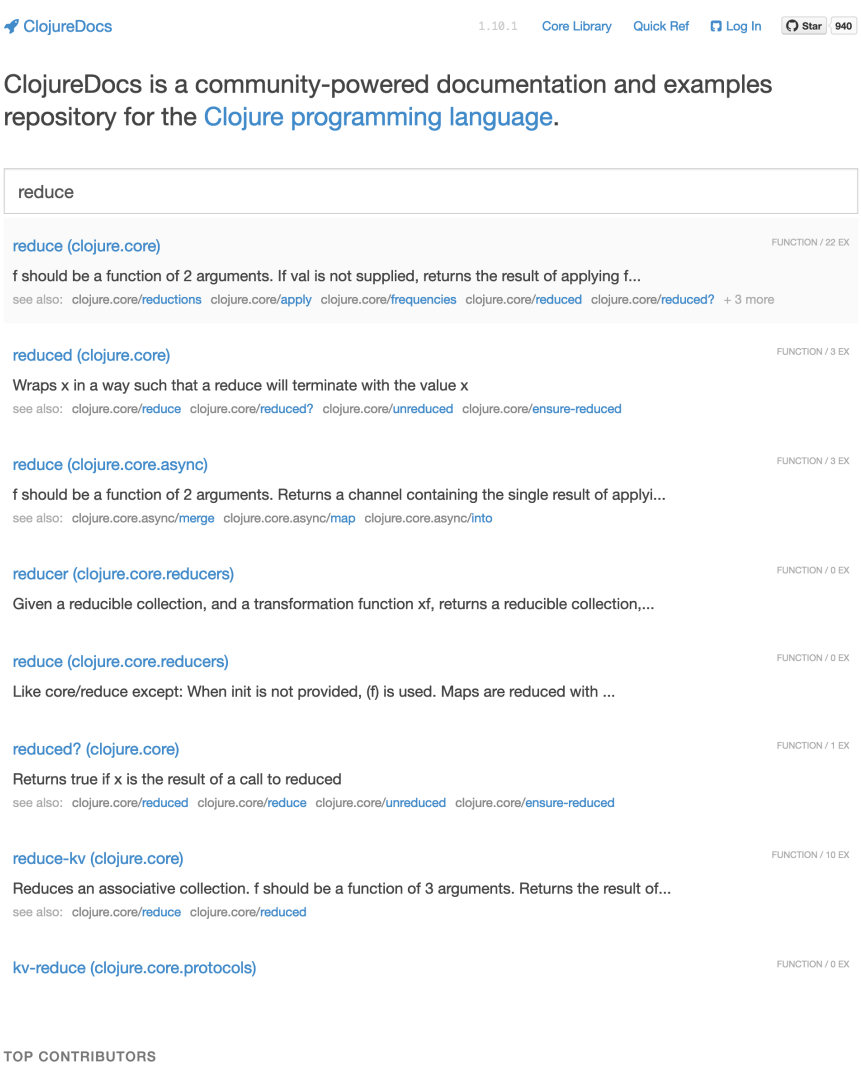
[1] https://en.wikipedia.org/wiki/Random-access_memory

[2] https://en.wikipedia.org/wiki/Immutable_object

Chapter 5. Reading Clojure Documentation

5.1. Clojure docs online


ClojureDocs
1.10.1
Core Library
Quick Ref
Log In
Star
940
ClojureDocs is a community-powered documentation and examples repository for the Clojure programming language.
Looking for?
TOP CONTRIBUTORS
Migrate your old ClojureDocs account
RECENTLY UPDATED
fsrc authored a note for clojure.core/constantly 5 hours ago.
djblue authored an example for clojure.pprint/pprint 3 days ago.
wactbprot added a see-also from clojure.core/remove-watch to clojure.core/agent 12 days ago.
Obscerno authored an example for clojure.core/try 13 days ago.
djblue added a see-also from clojure.java.shell/sh to clojure.core/shutdown-agents 15 days ago.
djblue authored an example for clojure.core/add-tap 15 days ago.
ON CLOJURE
Clojure is a concise, powerful, and performant general-purpose programming language that runs on the JVM, CLR, Node.js, and modern mobile and desktop web browsers.
New to Clojure and not sure where to start? Here are a few good resources to get you off on the right foot:
Rich Hickey's Greatest Hits (videos)
Clojure for the Brave and True
Clojure from the Ground Up
4Clojure (learn Clojure interactively)
ClojureScript Koans
Run Clojure code live in your browser
There's no denying that Clojure is just so *different* from
;; Let's define some data using list / map
;; literals:
(def scenes [{:subject "Frankie"
:action "say"
:object "relax"}
{:subject "Lucy"
:action "❤s"
:object "Clojure"}
{:subject "Rich"
:action "tries"
:object "a new conditioner"}])
;; Define a function
(defn people-in-scenes [scenes]
(->> scenes
(map :subject)
(interpose ", ")


ClojureDocs
1.10.1
Core Library
Quick Ref
Log In
Star 940
ClojureDocs is a community-powered documentation and examples repository for the Clojure programming language.
reduce
reduce (clojure.core)
FUNCTION / 22 EX
f should be a function of 2 arguments. If val is not supplied, returns the result of applying f...
see also: clojure.core/reductions clojure.core/apply clojure.core/frequencies clojure.core/reduced clojure.core/reduced? + 3 more
reduced (clojure.core)
FUNCTION / 3 EX
Wraps x in a way such that a reduce will terminate with the value x
see also: clojure.core/reduce clojure.core/reduced? clojure.core/unreduced clojure.core/ensure-reduced
reduce (clojure.core.async)
FUNCTION / 3 EX
f should be a function of 2 arguments. Returns a channel containing the single result of applyi...
see also: clojure.core.async/merge clojure.core.async/map clojure.core.async/into
reducer (clojure.core.reducers)
FUNCTION / 0 EX
Given a reducible collection, and a transformation function xf, returns a reducible collection,...
reduce (clojure.core.reducers)
FUNCTION / 0 EX
Like core/reduce except: When init is not provided, (f) is used. Maps are reduced with ...
reduced? (clojure.core)
FUNCTION / 1 EX
Returns true if x is the result of a call to reduced
see also: clojure.core/reduced clojure.core/reduce clojure.core/unreduced clojure.core/ensure-reduced
reduce-kv (clojure.core)
FUNCTION / 10 EX
Reduces an associative collection. f should be a function of 3 arguments. Returns the result of...
see also: clojure.core/reduce clojure.core/reduced
kv-reduce (clojure.core.protocols)
FUNCTION / 0 EX
TOP CONTRIBUTORS

NAV

Top
Examples 22
See Also 8
Notes 3

NAMESPACES

clojure
core
async
logic
fd
pldb
protocols
reducers
server
data
datafy
edn
inspector
instant
java
browse
io
javadoc
shell
main
pprint
reflect
repl
set
spec
alpha
stacktrace
string
template
test
junit
tap
walk
xml
zip

reduce

clojure.core

Available since 1.0 (source)

```
(reduce f coll)  (reduce f val coll)
```

```
f should be a function of 2 arguments. If val is not supplied,
returns the result of applying f to the first 2 items in coll, then
applying f to that result and the 3rd item, etc. If coll contains no
items, f must accept no arguments as well, and reduce returns the
result of calling f with no arguments.  If coll has only 1 item, it
is returned and f is not called.  If val is supplied, returns the
result of applying f to val and the first item in coll, then
applying f to that result and the 2nd item, etc. If coll contains no
items, returns val and f is not called.
```

© Rich Hickey. All rights reserved. Eclipse Public License 1.0

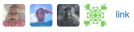
22 EXAMPLES

link

```
(reduce + [1 2 3 4 5])  ;;=> 15
(reduce + [])           ;;=> 0
(reduce + [1])          ;;=> 1
(reduce + [1 2])        ;;=> 3
(reduce + 1 [])         ;;=> 1
(reduce + 1 [2 3])      ;;=> 6
```

link

```
;; Converting a vector to a set:

(reduce conj #{} [:a :b :c])
;; => #{:a :c :b}
```

 link

```
;; Create a word frequency map out of a large string s.

;; `s` is a long string containing a lot of words :)
(reduce #(assoc %1 %2 (inc (%1 %2 0)))
        {}
        (re-seq #"\w+" s))

; (This can also be done using the `frequencies` function.)
```

Chapter 6. Logic and comparison

https://gitlab.com/clojure-book/code/-/raw/master/logic-and-comparison.clj

```
(= 1 1)
```

Output

```
true
```

```
(= 1 2)
```

Output

```
false
```

```
(= 7 7 7 )
```

Output

```
true
```

```
(= 7 8 7)
```

Output

```
false
```

```
(> 1 2)
```

Output

```
false
```

```
(> 2 1)
```

Output

```
true
```

```
(> 3 2 1)
```

Output

```
true
```

```
(>= 5 5)
```

Output

```
true
```

```
(>= 6 5)
```

Output

```
false
```

```
(>= 6 7)
```

Output

```
false
```

```
(< 1 2)
```

Output

```
true
```

```
(< 1 2 3)
```

Output

```
true
```

```
(< 2 1)
```

Output

```
false
```

```
(< 1 3 2)
```

Output

```
false
```

```
(<= 7 7)
```

Output

```
true
```

```
(<= 7 8)
```

Output

```
false
```

```
(<= 8 7)
```

Output

```
false
```

```
(and true true)
```

Output

```
true
```

```
(and  true false)
```

Output

```
false
```

```
(or true true)
```

Output

```
true
```

```
(or true false)
```

Output

```
true
```

```
(or false false)
```

Output

```
false
```

```
(not true)
```

Output

```
false
```

```
(not false)
```

Output

```
true
```

Chapter 7. Conditions and branching

7.1. if

```
;; if.clj

(let [a 5]
  (if (> a 0)
    (println a "is positive")))
```

```
;; if_else.clj

(let [a -5]
  (if (> a 0)
    (println a "is positive")
    (println a "is negative")))
```

```
;; if_nested.clj

(let [a 5]
  (if (> a 0)
    (println a "is positive")
    (if (< a 0)
      (println a "is negative")
      (println a "is neither positive nor negative"))))
```

```
;; if_multiple_statemnts_in_branch.clj

(let [a 5]
  (if (> a 0)

    (do
      (println a "is negative")
      (println "There are infinite positive numbers"))

    (do
      (println a "is negative")
      (println "There are infinite negative numbers"))))
```

7.2. when

```
;; when.clj
```

```
(let [print-something true]
  (when print-something
    (println "I print something.")
    (println "I print other things too.")))
```

7.3. cond

```
;; cond.clj

(let [number 5]
  (cond
    (> number 0) (println number "is positive.")
    (< number 0) (println number "is negative.")
    :else (println number "is neither positive nor negative.")))
```

```
(let [number 5]
  (cond
    (> number 0) (println number "is positive.")
    (< number 0) (println number "is negative.")))
```

Chapter 8. Functions

```
;; function_say_hello.clj

(defn say-hello []
  (println "Hello world!"))

(say-hello)
```

```
;; function_with_arguments.clj

(defn say-hello [name]
  (println "Hello" name "!"))

(say-hello "Karthik")
```

```
;; function-multiple-arguments.clj

(defn multiple-args
  ([arg-1] (println "One argument passed:" arg-1))
  ([arg-1 arg-2] (println "Two argument passed:" arg-1 arg-2)))

(multiple-args 1)
(multiple-args 1 2)
```

Output

```
One argument passed: 1
Two argument passed: 1 2
```

```
;; function_unlimited_arguments.clj

(defn unlimited-arguments [& args]
  (println "Arguments:" args)
  (println "Type of args:" (type args)))

(unlimited-arguments 1)
(println)
(unlimited-arguments 1 17 true "Karthik" :coder)
```

Output

```
Arguments: (1)
```

```
Type of args: clojure.lang.ArraySeq

Arguments: (1 17 true Karthik :coder)
Type of args: clojure.lang.ArraySeq
```

```
;; function_unlimited_arguments_2.clj

(defn unlimited-arguments [first-arg & args]
  (println "First argument:" first-arg)
  (println "Other arguments:" args))

(unlimited-arguments 1)
(println)
(unlimited-arguments 1 17 true "Karthik" :coder)
```

Output

```
First argument: 1
Other arguments: nil

First argument: 1
Other arguments: (17 true Karthik :coder)
```

```
;; function_unlimited_arguments_3.clj

(defn unlimited-arguments [first-arg second-arg & args]
  (println "First argument:" first-arg)
  (println "Second argument:" second-arg)
  (println "Other arguments:" args))

(unlimited-arguments 1 2)
(println)
(unlimited-arguments 1 17 true "Karthik" :coder)
```

Output

```
First argument: 1
Second argument: 2
Other arguments: nil

First argument: 1
Second argument: 17
Other arguments: (true Karthik :coder)
```

```
;; function_returning_something.clj
```

```clojure
(defn add [a b]
  (+ a b))

(println (add 2  3))
```

8.1. Recursion

```clojure
;; function_recursion.clj

(defn count-down [number]
  (println number)
  (if (not (= 0 (dec number)))
    (count-down (dec number))))

(count-down 5)
```

```clojure
;; function_recur.clj

(defn count-down [number]
  (println number)
  (if (not (= 0 (dec number)))
    (recur (dec number))))

(count-down 5)
```

```clojure
;; function_sum_using_recursion.clj

(defn sum [numbers total]
  (if (empty? numbers)
    total
    (recur (rest numbers) (+ total (first numbers)))))

(println (sum [1 2 3 4 5], 0))
```

```clojure
;; function_collection_sum.clj

(defn sum [numbers total]
  (if (empty? numbers)
    total
    (recur (rest numbers) (+ total (first numbers)))))

(defn collection-sum [collection]
  (sum collection 0))
```

```
(println (collection-sum [1 2 3 4 5]))
```

8.2. Multimethods

```
;; without_multimethods.clj

(defn print-welcome-message [person]
  (cond
    (string? person) (println "Welcome" person)
    (vector? person) (println  "Welcome" (first person) "from" (last person))
    (map? person)    (println "Welcome" (person "name") "from" (person "from"))))

(print-welcome-message "Karthik from Chennai")
(print-welcome-message ["Kalam" "Ramanthapuram"])
(print-welcome-message {"name" "Bharathiyaar" "from" "Yettaiyapuram"})
```

Output

```
Welcome Karthik from Chennai
Welcome Kalam from Ramanthapuram
Welcome Bharathiyaar from Yettaiyapuram
```

```
;; with_multimethods.clj

(defn welcome-person [person]
  (cond
    (string? person) :welcome-person-string
    (vector? person) :welcome-person-vector
    (map? person)    :welcome-person-map))

(defmulti print-welcome-message welcome-person)

(defmethod print-welcome-message :welcome-person-string [person]
  (println "Welcome" person))

(defmethod print-welcome-message :welcome-person-vector [person]
  (println  "Welcome" (first person) "from" (last person)))

(defmethod print-welcome-message :welcome-person-map [person]
  (println "Welcome" (person "name") "from" (person "from")))

(print-welcome-message "Karthik from Chennai")
(print-welcome-message ["Kalam" "Ramanthapuram"])
(print-welcome-message {"name" "Bharathiyaar" "from" "Yettaiyapuram"})
```

Output

```
Welcome Karthik from Chennai
Welcome Kalam from Ramanthapuram
Welcome Bharathiyaar from Yettaiyapuram
```

8.3. Pre and Post Condition Checking

```
;; function_pre.clj

(defn sum [a b]
  {:pre [(number? a) (number? b)]}
  (+ a b))

(println (sum 4 5))
;; (println (sum "4" 5)) ;; Thows an error
```

```
;; function_post.clj

(defn sum [a b]
  {:post [(number? %)]}
  "45")

(defn sum-without-post [a b]
  "45")

(println (sum-without-post 4 5))
(println (sum 4 5)) ;; ; Assert failed: (number? %)
```

```
;; function_pre_post.clj

(defn sum [a b]
  {:pre [(number? a) (number? b)]
   :post [(number? %)]}
  (+ a b))

(println (sum 4 5))
```

8.4. Docstring

```
;; docstring.clj

(defn sum
  "Adds two numbers passed as arguments.

   The argumets should be numbers.
```

```
   **Usage**

   ```clojure
   (sum 4 5) ;; returns 9
   ```
  "
  [a b]
  {:pre [(number? a) (number? b)]}
  (+ a b))

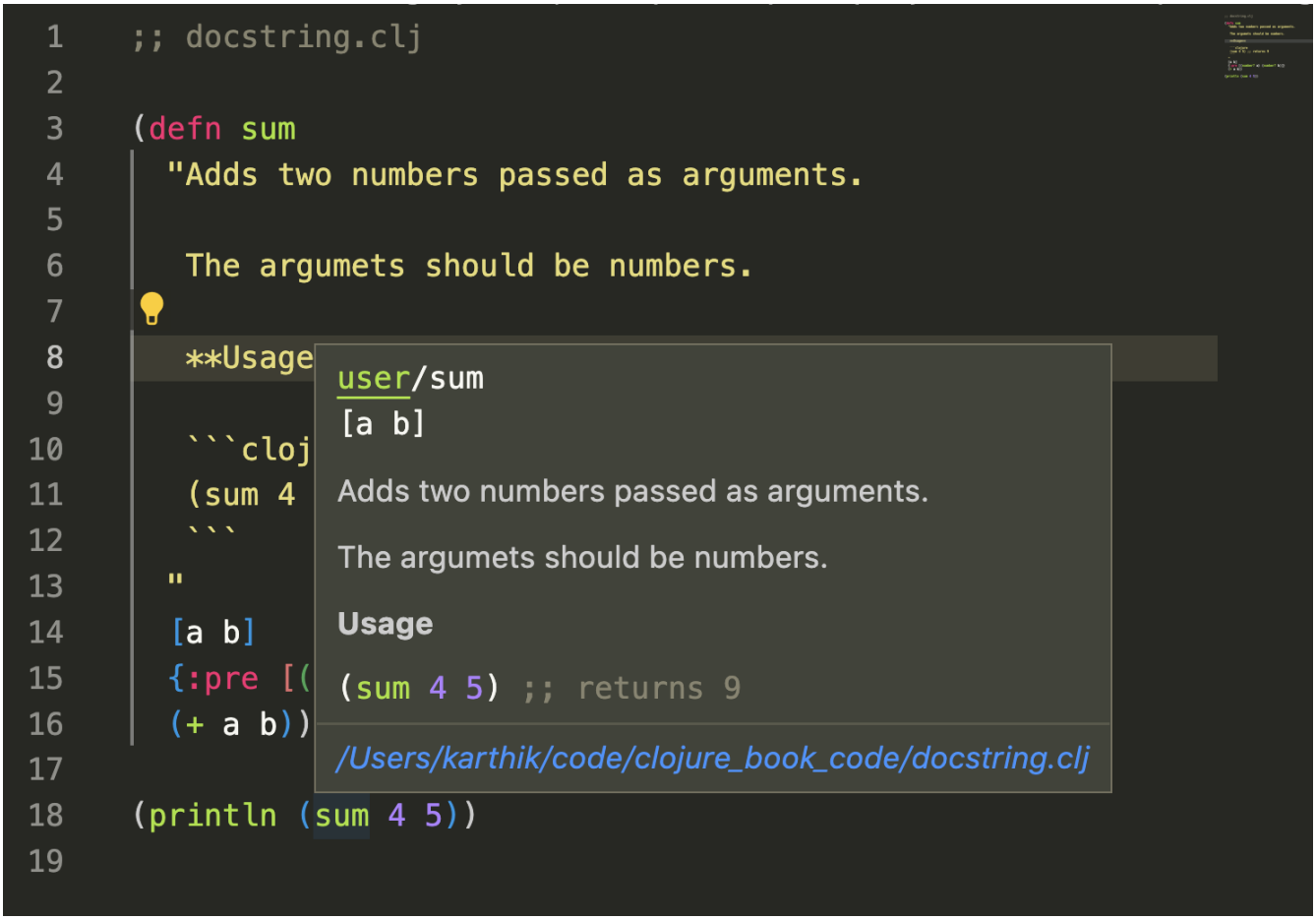
(println (sum 4 5))
```

```
clj□user□> (doc sum)
-------------------------
user/sum
([a b])
  Adds two numbers passed as arguments.

   The argumets should be numbers.

   **Usage**

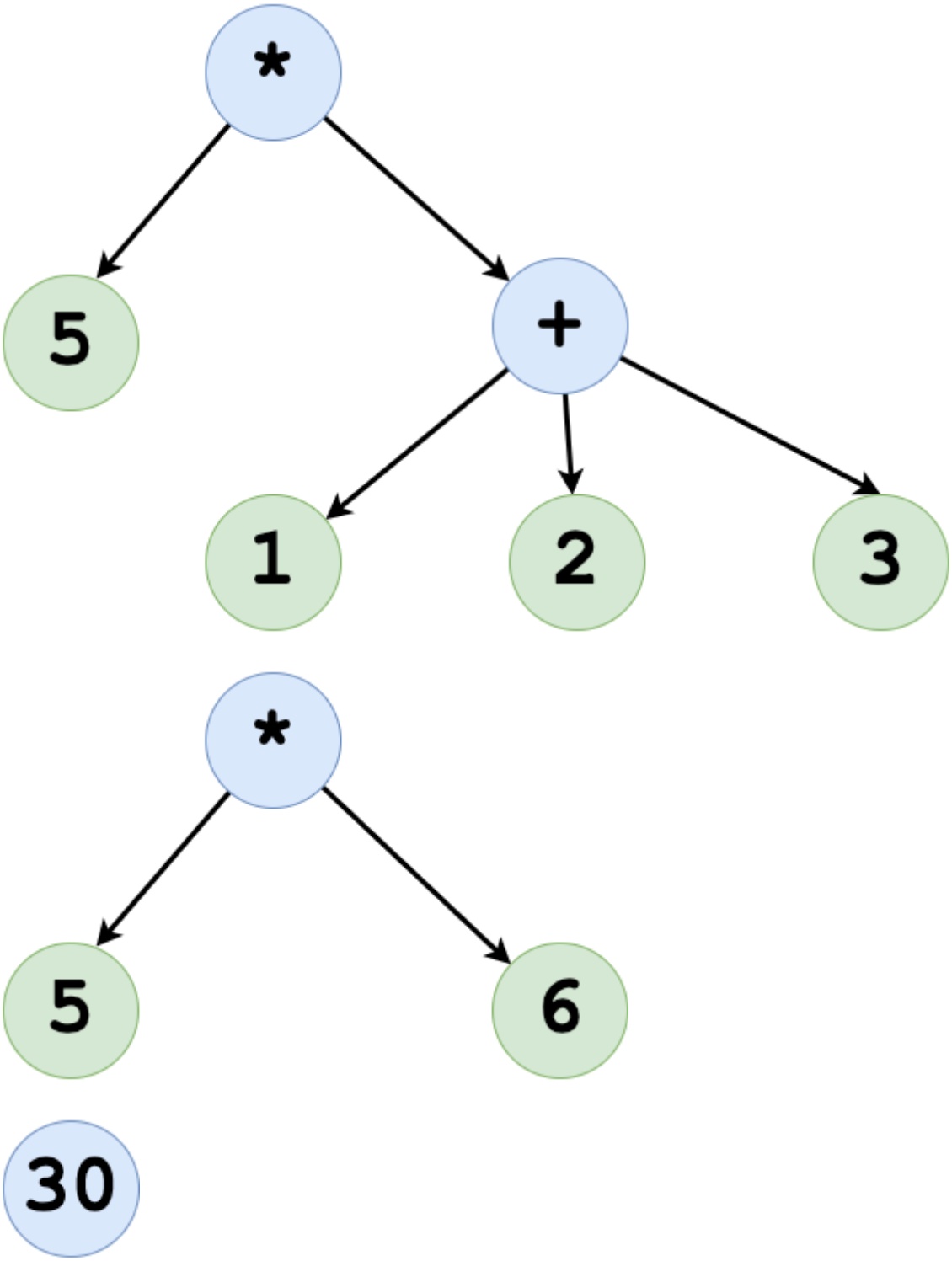
   ```clojure
   (sum 4 5) ;; returns 9
   ```


nil
```


;; docstring.clj
(defn sum
"Adds two numbers passed as arguments.
The argumets should be numbers.
**Usage
```cloj
(sum 4
```
"
[a b]
{:pre [(
(+ a b))
(println (sum 4 5))
user/sum
[a b]
Adds two numbers passed as arguments.
The argumets should be numbers.
Usage
(sum 4 5) ;; returns 9
/Users/karthik/code/clojure_book_code/docstring.clj

Chapter 9. What is Clojure code

```
(* 5 (+ 1 2 3))
```



Chapter 10. Sequence

Chapter 11. Lazy Sequence

11.1. repeat

```
(def multiverse (repeat "universe"))
```

```
(first multiverse)
```

```
(take 5 multiverse)
```

```
;; (rest multiverse) ;; doesn't seem to work
```

```
;; (last multiverse) ;; doesn't seem to work
```

```
(nth multiverse 10,000)
```

```
(nth multiverse 100)
```

11.2. cycle

```
(def multiverse (cycle ["universe" "antiverse"]))
```

```
(first multiverse)
```

```
(second multiverse)
```

```
(take 5 multiverse)
```

```
(nth multiverse 2000)
```

```
(nth multiverse 1783)
```

```
(def tiffin-items
  (cycle ["idli", "vadai", "dosai", "sambar"]))
```

```
(nth tiffin-items 5)
```

```
(take 10 tiffin-items)
```

11.3. iterate

```
(def all-numbers (iterate inc 1))
```

```
(nth all-numbers 24)
```

```
(take 10 all-numbers)
```

Chapter 12. map, filter, reduce

12.1. map

```
(map inc [1 2 3])
```

```
(map (fn [x] (+ x 1)) [1 2 3])
```

```
(map #(+ % 1) [1 2 3])
```

```
(map (fn [x y]
       (+ (Math/pow x 2)
          (Math/pow y 2))) [1 2 3] [2 3 4])
```

```
(defn double-it [x]
  (* x 2))

(map double-it [1 2 3])
```

12.2. filter

```
(filter odd? (range 1 11))
```

```
(filter even? (range 1 11))
```

```
(filter #(> % 5) (range 1 11))
```

```
(filter #(> 5 %) (range 1 11))
```

```
(filter (fn [x] (> x 5)) (range 1 11))
```

```
(defn greater-than-5 [x]
  (> x 5))
```

```
(filter greater-than-5 (range 1 11))
```

12.3. reduce

```
(reduce + [1 2 3 4])
```

```
(reduce * [1 2 3 4])
```

```
(defn add [a b]
  (+ a b))
```

```
(reduce add [1 2 3 4])
```

```
(defn multiply [a b]
  (* a b))
```

```
(reduce multiply [1 2 3 4])
```

Chapter 13. Destructuring

```
;; vector_destructuring.clj

(def people ["Rehmaan" "Kalaam"])

(let [[musician scientist] people]
  (println "Musician is" musician)
  (println "Scientist is" scientist))
```

Output

```
Musician is Rehmaan
Scientist is Kalaam
nil
```

```
;; vector_destructuring_2.clj

(def people ["Rehmaan" "Kalaam"])

(let [[musician scientist artist] people]
  (println "Musician is" musician)
  (println "Scientist is" scientist)
  (println "Artist is" artist))
```

Output

```
Musician is Rehmaan
Scientist is Kalaam
Artist is nil
nil
```

```
;; vector_destructuring_3.clj

(def people ["Rehmaan" "Kalaam" "Hussein" "Madhavan"])

(let [[musician scientist artist] people]
  (println "Musician is" musician)
  (println "Scientist is" scientist)
  (println "Artist is" artist))
```

Output

```
Musician is Rehmaan
Scientist is Kalaam
Artist is Hussein
nil
```

```
;; vector_destructuring_4.clj

(def people ["Rehmaan" "Kalaam" "Hussein" "Madhavan"])

(let [[musician scientist artist actor] people]
  (println "Musician is" musician)
  (println "Scientist is" scientist)
  (println "Artist is" artist)
  (println "Actor is" actor))
```

Output

```
Musician is Rehmaan
Scientist is Kalaam
Artist is Hussein
Actor is Madhavan
nil
```

```
;; vector_destructuring_5.clj

(def people ["Rehmaan" "Kalaam" "Hussein" "Madhavan"])

(let [[_ scientist _ actor] people]
  (println "Scientist is" scientist)
  (println "Actor is" actor))
```

Output

```
Scientist is Kalaam
Actor is Madhavan
nil
```

```
;; map_destructuring_4.clj

(def people
  {:musician "Rehmaan"
   :scientist "Kalaam"
   :artist "Hussein"
   :actor "Madhavan"})
```

```
(let [{scientist :scientist actor :actor} people]
  (println "Scientist is" scientist)
  (println "Actor is" actor))
```

Output

```
Scientist is Kalaam
Actor is Madhavan
nil
```

Bibliography

- Clojure for the brave and true https://www.braveclojure.com/
- Getting Clojure https://amzn.to/3ABmVup
- Programming Clojure https://amzn.to/3c2aAFe
- Professional Clojure https://amzn.to/3ARddo7

Many hackers swear by LISP, and just like any other programmer coming from OOPS background I was skeptical about functional programming. When I started learning it, it wasn't straight forward (for an OOP developer), but I persisted and when I reached what's called REPL driven development I was blown away.

Clojure is a dialect of LISP, and it was smartly designed to be hosted and not compiled, which means that when you learn it, you can target multiple platforms like Java runtime, Common Language Runtime, JavaScript and Dart platforms, hence your job potential will be huge if you master it.

Clojure and all LISP based languages, surprisingly has only one syntax, which makes it simple and unimaginably powerful, and hence makes it easier to learn compared to other languages, and hence I feel it could be great first programming language for many.

Join me in this wonderful journey to learn a dialect of a language which refuses to die and rises like phoenix.

- Karthikeyan A K